Chapter 3 – Describing Syntax and Semantics

CS-4337 Organization of Programming Languages

Dr. Chris Irwin Davis

Email: cid021000@utdallas.edu
Phone: (972) 883-3574
Office: ECSS 4.705

Chapter 3 Topics

- Introduction
- The General Problem of Describing Syntax
- Formal Methods of Describing Syntax
- Attribute Grammars
- Describing the Meanings of Programs: Dynamic Semantics

Introduction

- Syntax: the form or structure of the expressions, statements, and program units
- Semantics: the meaning of the expressions, statements, and program units
- Syntax and semantics provide a language's definition
  - Users of a language definition
    - Other language designers
    - Implementers
    - Programmers (the users of the language)

The General Problem of Describing Syntax: Terminology

- A sentence is a string of characters over some alphabet
- A language is a set of sentences
- A lexeme is the lowest level syntactic unit of a language (e.g., `*`, `sum`, `begin`)
- A token is a category of lexemes (e.g., identifier)

Example: Lexemes and Tokens

```
index = 2 * count + 17
```

Lexemes	*Tokens*
`index`	`identifier`
`=`	`equal_sign`
`2`	`int_literal`
`*`	`mult_op`
`count`	`identifier`
`+`	`plus_op`
`17`	`int_literal`
`;`	`semicolon`

Formal Definition of Languages

- Recognizers
  - A recognition device reads input strings over the alphabet of the language and decides whether the input strings belong to the language
  - Example: syntax analysis part of a compiler
    - Detailed discussion of syntax analysis appears in Chapter 4
- Generators
  - A device that generates sentences of a language
  - One can determine if the syntax of a particular sentence is syntactically correct by comparing it to the structure of the generator

Formal Methods of Describing Syntax

- Formal language-generation mechanisms, usually called **grammars**, are commonly used to describe the syntax of programming languages.

BNF and Context–Free Grammars

- Context-Free Grammars
  - Developed by Noam Chomsky in the mid-1950s
  - Language generators, meant to describe the syntax of natural languages
  - Define a class of languages called context-free languages
- Backus-Naur Form (1959)
  - Invented by John Backus to describe the syntax of Algol 58
  - BNF is equivalent to context-free grammars

BNF Fundamentals

- In BNF, abstractions are used to represent classes of syntactic structures — they act like syntactic variables (also called non-terminal symbols, or just non-terminals)

- Terminals are lexemes or tokens

- A rule has a left-hand side (LHS), which is a nonterminal, and a right-hand side (RHS), which is a string of terminals and/or nonterminals

BNF Fundamentals (continued)

- Nonterminals are often enclosed in angle brackets
  - Examples of BNF rules:

```
<ident_list> → identifier | identifier, <ident_list>
<if_stmt> → if <logic_expr> then <stmt>
```

- Grammar: a finite non-empty set of rules
- A start symbol is a special element of the nonterminals of a grammar

BNF Rules

- An abstraction (or nonterminal symbol) can have more than one RHS

```
<stmt> → <single_stmt>
               | begin <stmt_list> end
```

- The same as…

```
<stmt> → <single_stmt>
<stmt> → begin <stmt_list> end
```

Describing Lists

- Syntactic lists are described using recursion

```
<ident_list> → ident
             | ident, <ident_list>
```

- A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

An Example Grammar

```
<program> → <stmts>
<stmts> → <stmt> | <stmt> ; <stmts>
<stmt> → <var> = <expr>
<var> → a | b | c | d
<expr> → <term> + <term> | <term> - <term>
<term> → <var> | const
```

An Example Derivation

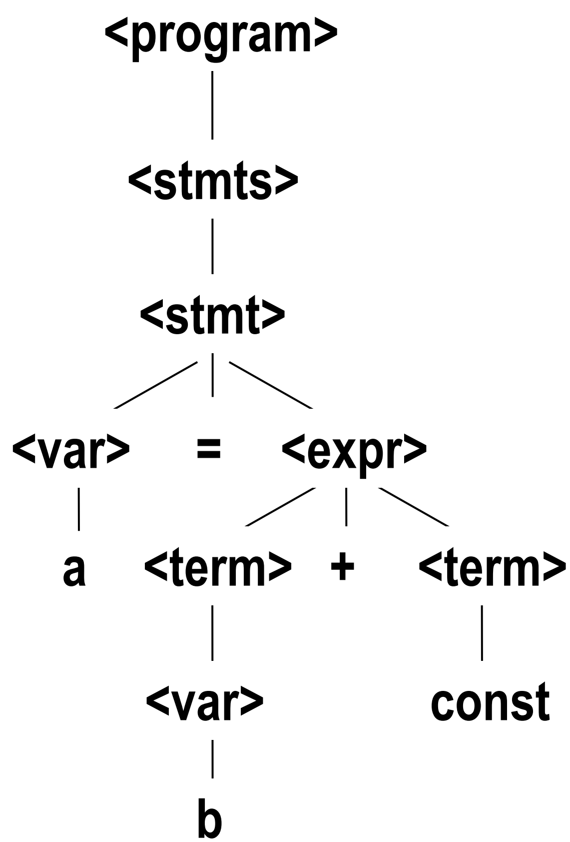
```
<program> => <stmts>
          => <stmt>
          => <var> = <expr>
          => a = <expr>
          => a = <term> + <term>
          => a = <var> + <term>
          => a = b + <term>
          => a = b + const
```

Derivations

- Every string of symbols in a derivation is a sentential form
- A sentence is a sentential form that has only terminal symbols
- A leftmost derivation is one in which the leftmost nonterminal in each sentential form is the one that is expanded
- A derivation may be neither leftmost nor rightmost

Parse Tree

- A hierarchical representation of a derivation



a = b + const

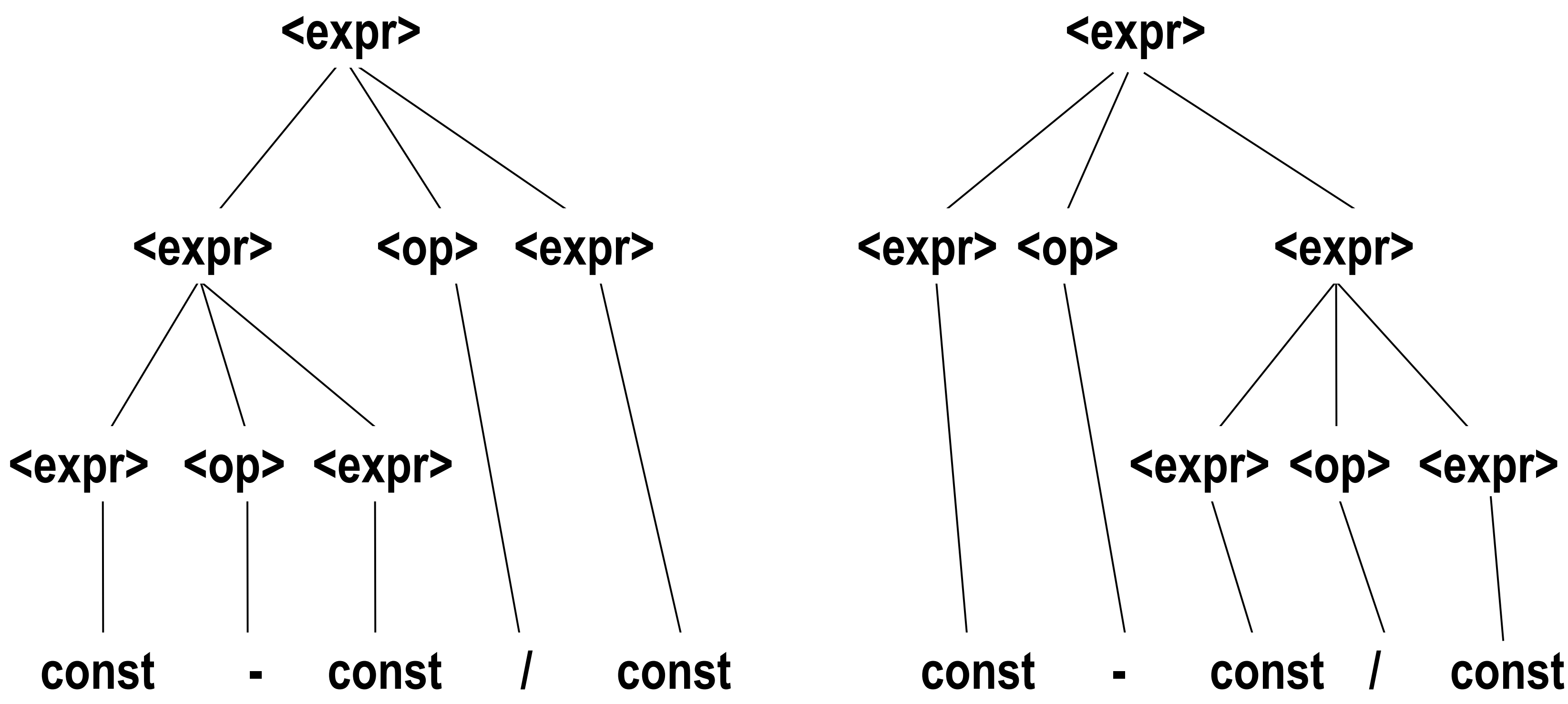
Ambiguity in Grammars

- A grammar is ambiguous if and only if it generates a sentential form that has two or more distinct parse trees

An Ambiguous Expression Grammar

<expr> → <expr> <op> <expr> | **const**

<op> → **/** | **-**

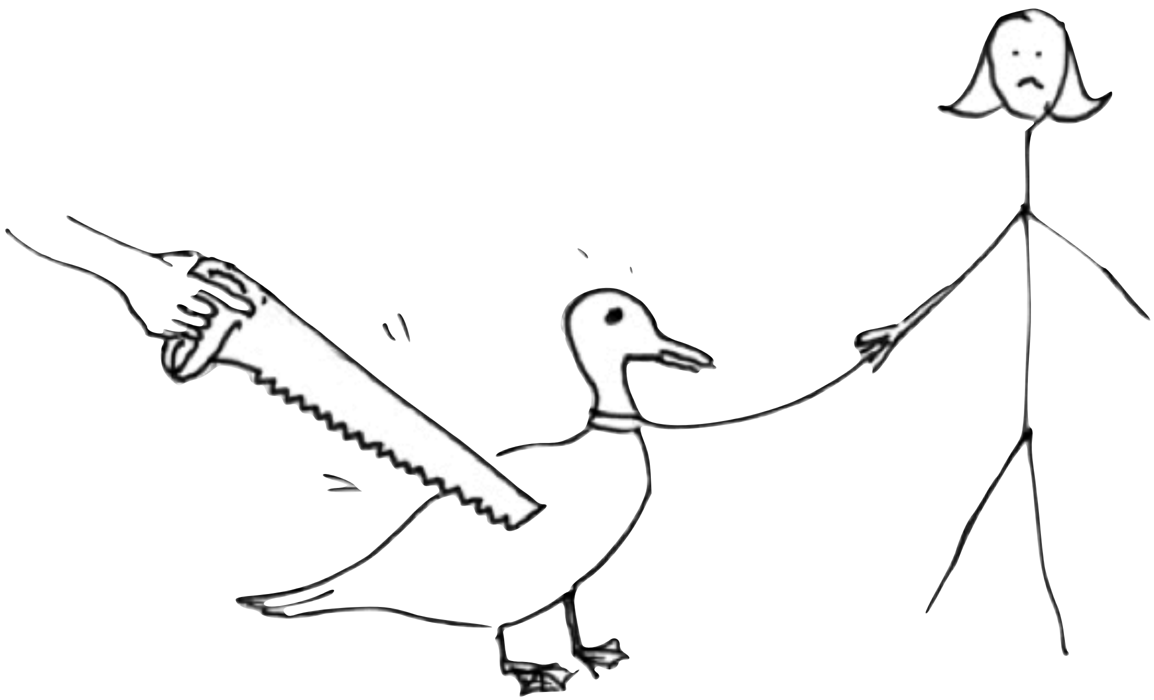


Ambiguous Grammars

- “I saw her duck”

Ambiguous Grammars

- "I saw her duck"



Ambiguous Grammars

"The men saw a boy in the park with a telescope"

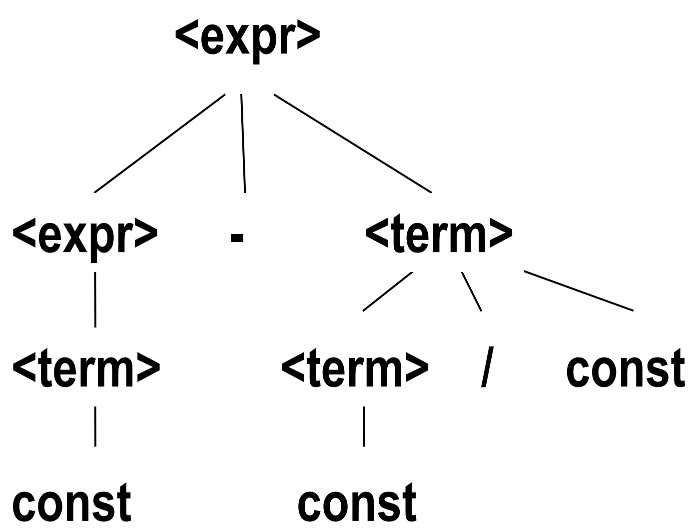
Logical Languages

- LOGLAN (1955)
  - Grammar based on predicate logic
  - Developed Dr. James Cooke Brown with the goal of making a language so different from natural languages that people learning it would think in a different way if the hypothesis were true
  - Loglan is the first among, and the main inspiration for, the languages known as logical languages, which also includes **Lojban** and **Ceqli**.
  - To invesitigate the Sapir-Whorf Hypothesis

An Unambiguous Expression Grammar

- If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity

<expr> → <expr> - <term> | <term>
<term> → <term> / **const**| **const**



Operator Precedence

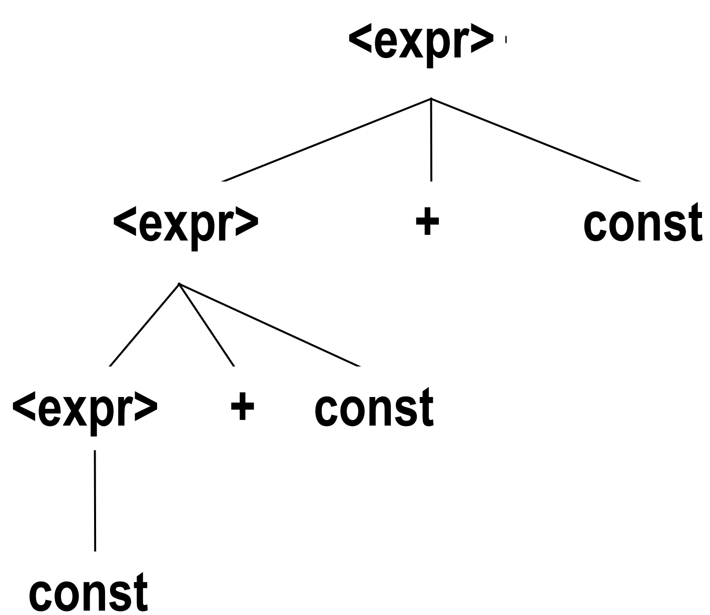
- If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity

<assign> → <id> = <expr>
<id> → **A** | **B** | **C**
<expr> → <expr> + <term> | <term>
<term> → <term> * <factor> | <factor>
<factor> → (<expr>) | <id>

Associativity of Operators

- Operator associativity can also be indicated by a grammar

```
<expr> -> <expr> + <expr> |  const  (ambiguous)
<expr> -> <expr> + const  |  const  (unambiguous)
```



Extended BNF

- Optional parts are placed in brackets [ ]

  `<proc_call> → ident [(<expr_list>)]`

- Alternative parts of RHSs are fplaced inside parentheses and separated via vertical bars

  `<term> → <term> (+|-) const`

- Repetitions (0 or more) are placed inside braces { }

  `<ident_list> → <identifier> {, <identifier>}`

BNF and EBNF

- BNF

```
<expr> → <term> |
               <expr> + <term> |
               <expr> - <term>
<term> → <factor> |
               <term> * <factor> |
               <term> / <factor>
```

- EBNF

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
```

Recent Variations in EBNF

- Alternative RHSs are put on separate lines
- Use of a colon instead of `=>`
- Use of $_{\texttt{opt}}$ for optional parts
- Use of `oneof` for choices

Attribute Grammars

Static Semantics

- Nothing to do with meaning
- Context-free grammars (CFGs) cannot describe all of the syntax of programming languages
- Categories of constructs that are trouble:
  - Context-free, but cumbersome (e.g., types of operands in expressions)
  - Non-context-free (e.g., variables must be declared before they are used)

Attribute Grammars

- Attribute grammars (AGs) have additions to CFGs to carry some semantic info on parse tree nodes

- Primary value of AGs:
  - Static semantics specification
  - Compiler design (static semantics checking)

Attribute Grammars : Definition

- Def: An attribute grammar is a context-free grammar $G = (S, N, T, P)$ with the following additions:
  - For each grammar symbol x there is a set A(x) of attribute values
  - Each rule has a set of functions that define certain attributes of the nonterminals in the rule
  - Each rule has a (possibly empty) set of predicates to check for attribute consistency

Attribute Grammars: Definition

- Let $X_0 \rightarrow X_1 \ldots X_n$ be a rule
- Functions of the form $S(X_0) = f(A(X_1), \ldots , A(X_n))$ define synthesized attributes
- Functions of the form $I(X_j) = f(A(X_0), \ldots , A(X_n))$, for $i <= j <= n$, define inherited attributes
- Initially, there are intrinsic attributes on the leaves

Attribute Grammars: An Example

- Syntax rule:

```
<proc_def> → procedure <proc_name>[1]
<proc_body> end <proc_name>[2];
```

- Predicate:

```
<proc_name>[1]string == <proc_name>[2].string
```

Attribute Grammars: An Example

- Syntax

```
<assign> → <var> = <expr>
<expr>   → <var> + <var> | <var>
<var>    → A | B | C
```

- actual_type: synthesized for <var> and <expr>
- expected_type: inherited for <expr>

Attribute Grammar (continued)

- Syntax rule: <expr> → <var>[1] + <var>[2]

  Semantic rules:

  <expr>.actual_type ← <var>[1].actual_type

  Predicate:

  <var>[1].actual_type == <var>[2].actual_type

  <expr>.expected_type == <expr>.actual_type

- Syntax rule: <var> → id

  Semantic rule:

  <var>.actual_type ← lookup (<var>.string)

Attribute Grammars (continued)

- How are attribute values computed?
  - If all attributes were inherited, the tree could be decorated in top-down order.
  - If all attributes were synthesized, the tree could be decorated in bottom-up order.
  - In many cases, both kinds of attributes are used, and it is some combination of top-down and bottom-up that must be used.

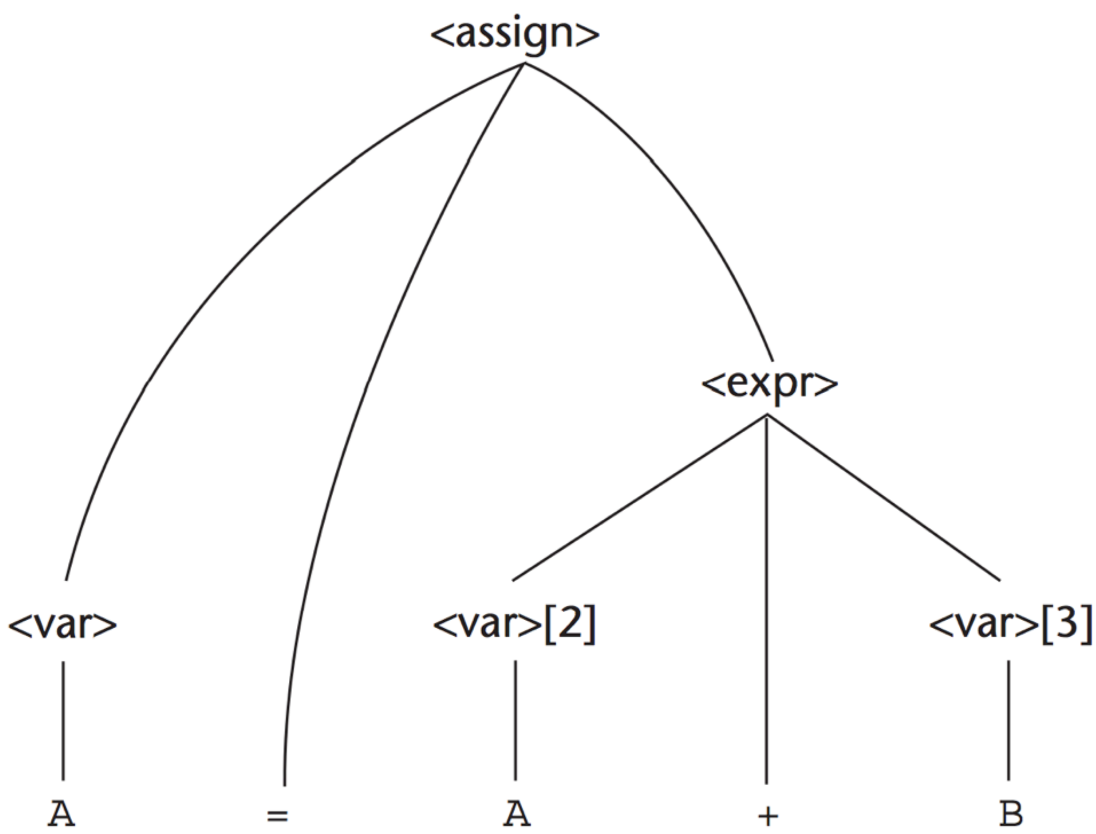
Attribute Grammars (continued)

```
<expr>.expected_type ← inherited from parent

<var>[1].actual_type ← lookup (A)
<var>[2].actual_type ← lookup (B)
<var>[1].actual_type =? <var>[2].actual_type

<expr>.actual_type ← <var>[1].actual_type
<expr>.actual_type =? <expr>.expected_type
```

Parse Tree



Computing Attribute Values

1. <var>.actual_type ← look-up(A) (Rule 4)

2. <expr>.expected_type ← <var>.actual_type (Rule 1)

3. <var>[2].actual_type ← look-up(A) (Rule 4)

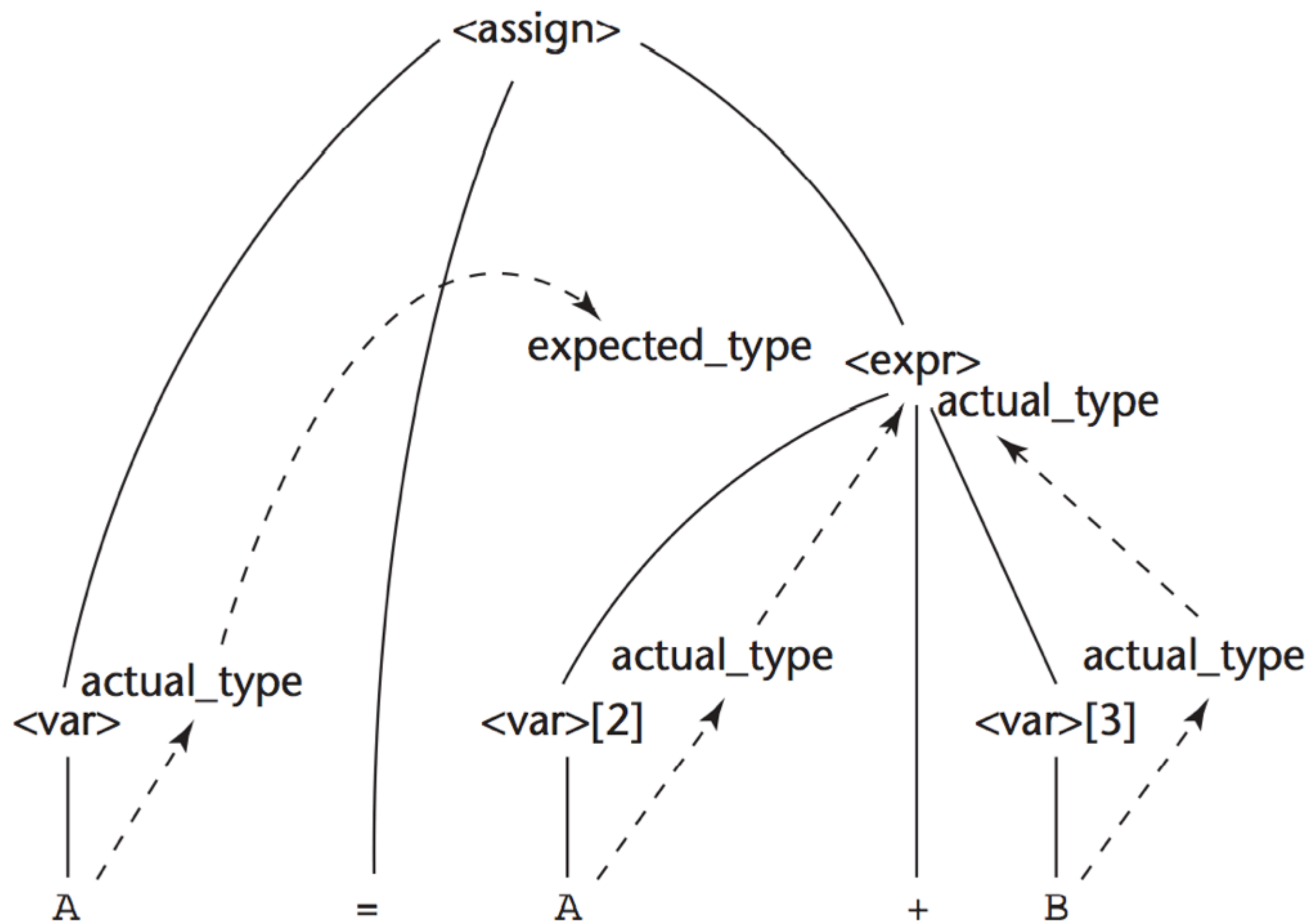
   <var>[3].actual_type ← look-up(B) (Rule 4)

4. <expr>.actual_type ← either int or real (Rule 2)

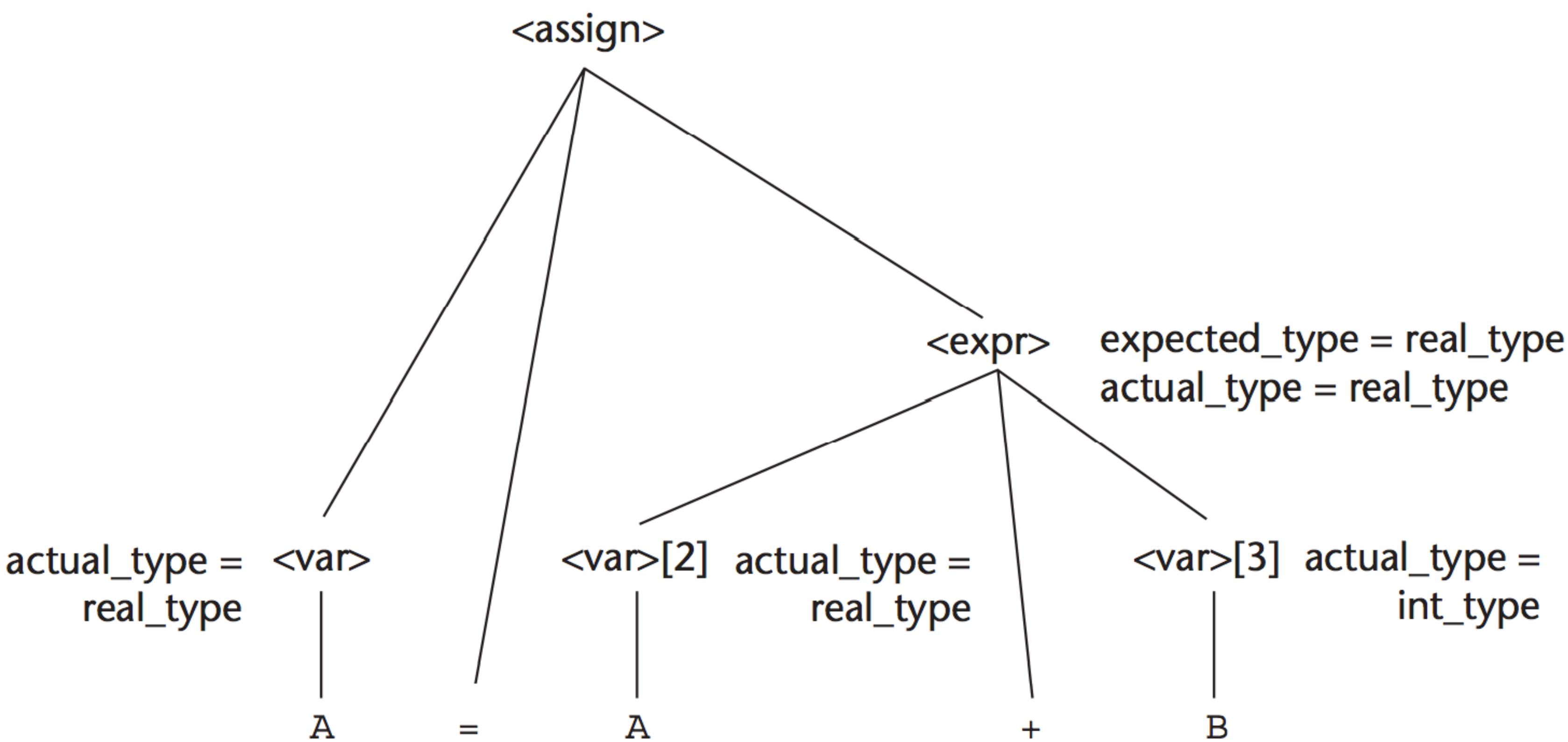
5. <expr>.expected_type == <expr>.actual_type is either

   TRUE or FALSE (Rule 2)

Flow of Attributes in the Tree



A Fully Attributed Parse Tree



Semantics

Semantics

- There is no single widely acceptable notation or formalism for describing semantics
- Several needs for a methodology and notation for semantics:
  - Programmers need to know what statements mean
  - Compiler writers must know exactly what language constructs do
  - Correctness proofs would be possible
  - Compiler generators would be possible
  - Designers could detect ambiguities and inconsistencies

Semantics

- Operational Semantics
- Denotational Semantics
- Axiomatic Semantics

Operational Semantics

- Operational Semantics
  - Describe the meaning of a program by executing its statements on a machine, either simulated or actual. The change in the state of the machine (memory, registers, etc.) defines the meaning of the statement
- To use operational semantics for a high-level language, a virtual machine is needed

Operational Semantics

- A hardware pure interpreter would be too expensive
- A software pure interpreter also has problems
  - The detailed characteristics of the particular computer would make actions difficult to understand
  - Such a semantic definition would be machine-dependent

Operational Semantics (continued)

- A better alternative: A complete computer simulation
- The process:
  - Build a translator (translates source code to the machine code of an idealized computer)
  - Build a simulator for the idealized computer
- Evaluation of operational semantics:
  - Good if used informally (language manuals, etc.)
  - Extremely complex if used formally (e.g., VDL), it was used for describing semantics of PL/I.

Operational Semantics (continued)

- Uses of operational semantics:
  - Language manuals and textbooks
  - Teaching programming languages
- Two different levels of uses of operational semantics:
  - Natural operational semantics
  - Structural operational semantics
- Evaluation
  - Good if used informally (language manuals, etc.)
  - Extremely complex if used formally (e.g.,VDL)

Denotational Semantics

- Based on recursive function theory
- The most abstract semantics description method
- Originally developed by Scott and Strachey (1970)

Denotational Semantics – continued

- The process of building a denotational specification for a language:
  - Define a mathematical object for each language entity
  - Define a function that maps instances of the language entities onto instances of the corresponding mathematical objects
- The meaning of language constructs are defined by only the values of the program's variables

Denotational Semantics: program state

- The state of a program is the values of all its current variables

$$s = \{<i_1, v_1>, <i_2, v_2>, \ldots, <i_n, v_n>\}$$

- Let **VARMAP** be a function that, when given a variable name and a state, returns the current value of the variable

$$\text{VARMAP}(i_j, s) = v_j$$

Evaluation of Denotational Semantics

- Can be used to prove the correctness of programs
- Provides a rigorous way to think about programs
- Can be an aid to language design
- Has been used in compiler generation systems
- Because of its complexity, it is of little use to language users

Axiomatic Semantics

- Based on formal logic (predicate calculus)
- Original purpose: formal program verification
- Axioms or inference rules are defined for each statement type in the language (to allow transformations of logic expressions into more formal logic expressions)
- The logic expressions are called assertions

Axiomatic Semantics (continued)

- An assertion before a statement (a precondition) states the relationships and constraints among variables that are true at that point in execution
- An assertion following a statement is a postcondition
- A weakest precondition is the least restrictive precondition that will guarantee the postcondition

Evaluation of Axiomatic Semantics

- Developing axioms or inference rules for all of the statements in a language is difficult
- It is a good tool for correctness proofs, and an excellent framework for reasoning about programs, but it is not as useful for language users and compiler writers
- Its usefulness in describing the meaning of a programming language is limited for language users or compiler writers

Denotation Semantics vs Operational Semantics

- In operational semantics, the state changes are defined by coded algorithms
- In denotational semantics, the state changes are defined by rigorous mathematical functions

Summary

- BNF and context-free grammars are equivalent meta-languages
  - Well-suited for describing the syntax of programming languages
- An attribute grammar is a descriptive formalism that can describe both the syntax and the semantics of a language
- Three primary methods of semantics description
  - Operation, Axiomatic, Denotational